Supporting the Flexible Federate

Christopher L. Martinez

Sterling Software
JNTFRDC – CSF 270
730 Irwin Avenue
Schriever AFB, CO 80912-7300
719-567-8546
chris.martinez@jntf.osd.mil

Keywords:
HLA, FOM, Wargame 2000.

ABSTRACT: *At the Joint National Test Facility, the Technology Insertion Studies and Analysis project implemented a prototype federation to reduce risk for Wargame 2000, a new command and control simulation. We found it difficult to flexibly specify and modify data interfaces among federates. An additional problem in our prototype development was federate visualization - we found it very difficult to show the workings of the federation. We recently worked with Wargame 2000 on a High Level Architecture gateway and were able to successfully resolve these two issues. Building on the lessons learned from our proto- federations, we extended our development tools by developing two innovative modules that use Thin Layer Data Interfacing. FedConnector is a generic framework that allowed rapid federate development. FedConsole is a federate viewer and control console. The two provide the developer access to federate data and federation components for runtime viewing and modification. FedConnector allowed data primitives and structures to be defined without recoding the simulation. Zero or more mappings were defined in the data and provided bi-directional data conversion. Applications are developed using these mappings with and without a Run Time Infrastructure connection and the development team has the choice between automatic FedConnector services or access the native RTI API. Through multiple mappings FedConnector supports interoperability between the federate and federation creating a flexible federate. FedConnector coupled with FedConsole facilitates federate/federation test and evaluation using a common benchmark. The two were used in the development of the War Game 2000 gateway as deliverable software modules. This project substantially decreased technical risk to modeling and simulation projects at the Joint National Test Facility that are required to be compliant with the High Level Architecture mandate.*

1. Introduction

Anyone who has been around the simulation community in the past few years has also been exposed to High Level Architecture (HLA). It has become apparent that HLA has brought new challenges that are currently being approached and solved in many ways. More specifically there is the challenge of Federation Object Model (FOM) dependence: when there are any changes to a Simulation Object Model (SOM) there will normally be a direct effect on the simulations FOM which typically leads to recoding. This paper describes the evolution of an HLA tool called FedConnector that has provided FOM independence for Wargame 2000 (WG2K) and Missile Defense Space Tool (MDST).

2. Approach

FedConnector started out as a small library designed to minimize the federate overhead involved in creating, joining and resigning from an HLA federation. The library succeeded in doing just that and made it easy for a federate to start/join/resign. The next logical step was to extend these services to some smart data types. A smart-data base class was developed and different data types could be created base on this data type. This idea became the basis for some of our early federates. The design suffered from

one major draw back: any changes in the FOM means changes to data names and types in the federate code.

About this time, WG2K had developed requirements for a configurable HLA Gateway. Since our work had a head start interfacing with the RTI API provided by the Defense Modeling and Simulation Offices, it was chosen as a starting point.

The object data for the Gateway is configured via a parameter file with HLA data via the SOM. The challenge was to devise a system that would allow changes to be made to either side. Our smart-data types were limited primarily because they made specific assumptions about data. We needed an architecture that started with primitive data types and allowed complex structures to be built from the primitives. This approach presented it's own challenges, it had to support nested structures and be fast. There was still one major obstacle to be addressed: a means of bridging differences in data names and types. These and other problems were solved through the development of a Thin Layer Data Interface (TLDI).

3. Thin Layer Data Interface

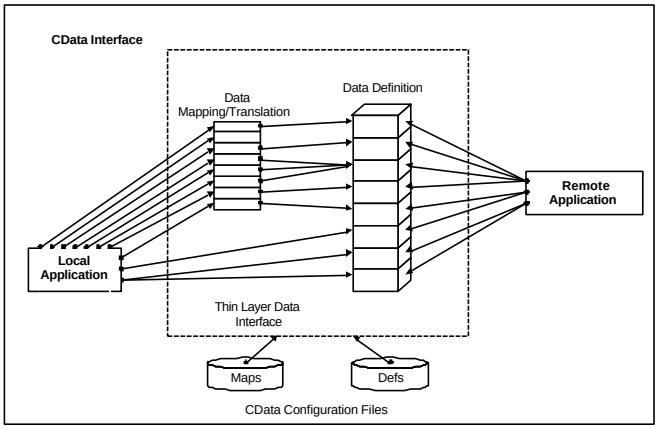

Figure 3.1 Thin Layer Data Interface De/Map

The bulk of the TLDI is pictured in Figure 3.1. It is shown configured by a definition and mapping file. Local and remote hosts operate sharing data that is defined by the data definition. The local host has the option of using the native format or one or more alternate-mapping interface. In the case of multiple mappings the TLDI notifies the local host of remote data creation/updates by calling the first update function registered.

The TLDI makes no assumptions about data and emphasizes speed and flexibility.

The TLDI supports user:

- Data Definition
- Data Mapping
- Data Conversion
- Data Instantiation
- Data Deletion
- Data Publication/Subscription
- Local/Remote Host Interfaces
- Local/Remote New/Updating/Deletion
- Data Constructor/Destructor
- Data Definition Associated User data
- Data Warehousing
- Data Utilities
- Definition Parser

1. Data Definition - Data primitives are defined using a name and a size. This scheme makes it possible to create a primitive data type with any name and size. Structures are defined based on data primitives as well as other structures. Structures can be nested to any level. These data definitions came to be known as "native data".

2. Data Mapping - Data mapping is accomplished by creating an alternate interface into the native data provided by the data definition. Multiple maps can be defined creating a many-to-one map-to-definition relationship. A map element can be associated directly with a native data element or it can reference an element of a nested structure.

3. Data Conversion - Data conversion is applied in a number ways. Normally it is defined in a map bridging a map element to a native element. Occasionally it will be assigned to a data primitive, as is the case with any sort of data pointer. There is a number built-in conversion functions that support most user requirements.

4. Data Instantiation - Based on the definition and map information the TLDI can instantiate data objects and provides all of the accessor and mutator functions.

5. Data Deletion - Data deletion is also handled based on the data definition. Since all data objects

are created dynamically, garbage collection is very straightforward.

6. Data Publication/Subscription – Data can be published and subscribed by type, map or element.

7. Local/Remote Host Interfaces - The TLDI provides a local interface that is used by the application and a remote interface that can be used by a gateway, or as you will see, an HLA interface. Both local and remote handles are maintained for both definition and instantiated data.

8. Local/Remote New/Updating/Deletion - Callbacks can be registered at the structure and element level for object creation, updates and deletion. This is the way the local application can be notified of the creation, change and deletion of a remote object.

9. Data Constructor/Destructor - Data constructors and destructor functions can be registered at the structure and element level to support specific initialization and clean up operations.

10. Data Definition Associated User data - There are two user definable integers and two void pointers that can be associated with any structure or structure element.

11. Data Warehousing - The onboard data warehouse maintains references to data where it is stored based on data type. Retrieval is based on the local or remote handle.

12. Data Utilities - The TLDI has utilities to support:

    - Documentation
    - Configuration Explanation
    - Definition Ambiguity

13. Definition Parser - Figure 3.2 shows the definition/mapping file format accepted by the TLDI built-in parser. The parser can be directed by the application to load a configuration file or it can be configured by setting the CDATA_DEF environmental variable.

    The TLDI parser supports the following functionality (numbers are keyed to Figure 3.2):

    1. Application Parameters
    2. Logging Definition Processing
    3. FedConsole Activation
    4. Data Primitive Definitions
    5. Conversion Function Assignments
    6. Data Definitions
    7. Map Definitions
    8. Publish and Subscribe
    9. User Data Assignment (integer)
    10. Utility Execution

```
//*****************************************************
// TestFed Definition File                  <FedConnecter>
//*****************************************************

// Application Parameters
FederationName   TestFederation
FederateName     TestFed
FedFileName      TestFed

// Open a file for outputing run
output: testFed.txt

// Turn FedConsole On
FedConsole On

// Data  Prinitive  Definitions
  int          4
  float        4
  double       8
  dataPtr      8      butoptr

// Assign Conversion Funcitons
  int          float        itof         ftoi
  int          double       itod         dtoi
  float        double       ftod         dtof

Struct     // Data Definitions
  position
    int x
    int y
    int z

  MunitionsEntiy
    int          id
    position     pos
    int          state

Message
    int type
    int size
    bytePtr data

MapStruct      // Map Definitions
  Missile MunitionsEntity
    float        missileId      id
    double       mX             pos.x
    double       mY             pos.y
    double       mZ             pos.z
    double       health         state

//Publish & Subscribe
Publish          Missile
Subscribe        Missile
Publish                         Message
Subscribe        Message

// User Definition data Assignments
Function         USER1          Missile.id     8192
Function         USER2          Missile.mX     20
Function         USER1          Missile.mY     55
Function         USER1          Missile.mZ     89

// Run Utilities
Function   buildDefs             // Build all
Function   dumpCStruct           // Output Defs in C format
Function   dumpFuncs             // Output Conv. Func. List
Function   dumpMaps              // Dump all Def Maps
Function   dumpConvAssign        // Output Conv. Func. Assignments
Function   checkNames            // Check for errors in defs

//*****************************************************
// End of testFed Definition File          <FedConnector>
```

Figure 3.2 TLDI
Definition/Mapping/Configuration

3.1 Using the TLDI

The are two C++ classes associated with the TLDI CDTypes and CData. CDTypes maintains all definition and configuration, while CData represents all instance data. A CDTypes class is instantiated and the developer loads configuration data either from a definition file using the built in parser, from the application, or a combination of the two. Update and delete functions are registered and the TLDI is ready for use by the local host. Data is instantiated by the CDTypes class in the form of CData objects. Accessor and mutator functions are available for all elements as well as several copy and assignment functions/operators.

The CDTypes object is now ready to be passed to a remote host for connection and configuration.

4. FedConnector

The target for the TLDI remote host was an HLA interface. This meant reusing the original small library and adapting it for use as a remote host. This library became the basis for FedConnector. FedConnector, reads the *federation .fed* file and combines the configuration information of the TLDI with the FOM reporting any errors found. All new/update/delete functions are registered with the TLDI to automate the processing of all-local objects and interactions. FedConnector then joins the federation and processes all publishing and subscription requirements.

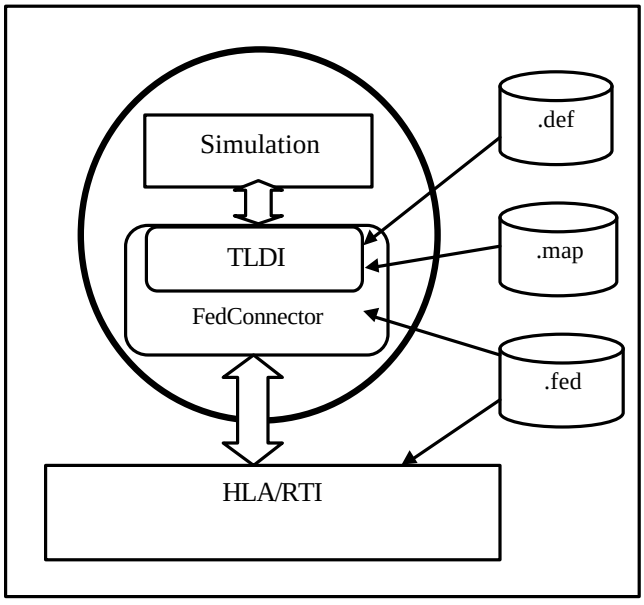

Figure4.1 Federate using FedConnector

Figure 4.1 shows a typical FedConnector based federate. Definition and map information are parsed by the TLDI. FedConnector parses and correlates FOM information.

FedConnector uses a single C++ class that contains the complete functionality of a remote HLA host as shown in Figure 4.1.

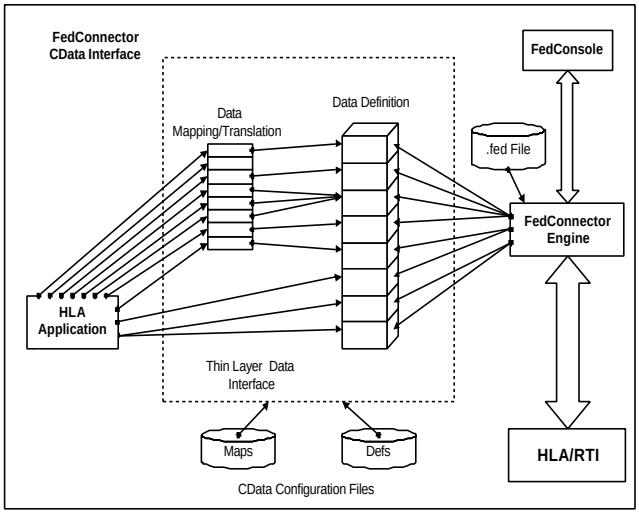

Figure 4.2 FedConnector as a Remote Host

5. FedConsole

A problem encountered during federate development was the lack of visibility. Out of this necessity came FedConsole. FedConsole is a Java application that provides a federation view from the federate perspective as shown in Figure 4.2.

The main screen is shown in Figure 5.1. There is a section for object, federation and interaction information. A current status section and a notes section provide the state of the federation and RTI.

The developer takes advantage of the Start/Stop/Pause/Reset buttons and well as 5 different sync/event buttons for federate control. Time management controls are available providing the 4 modes of time management. The developer passes

application information to the info, notes and status areas.

An object/interaction viewer is started from the main console. This allows objects/interactions listings, instance listings and object/interaction instance data.

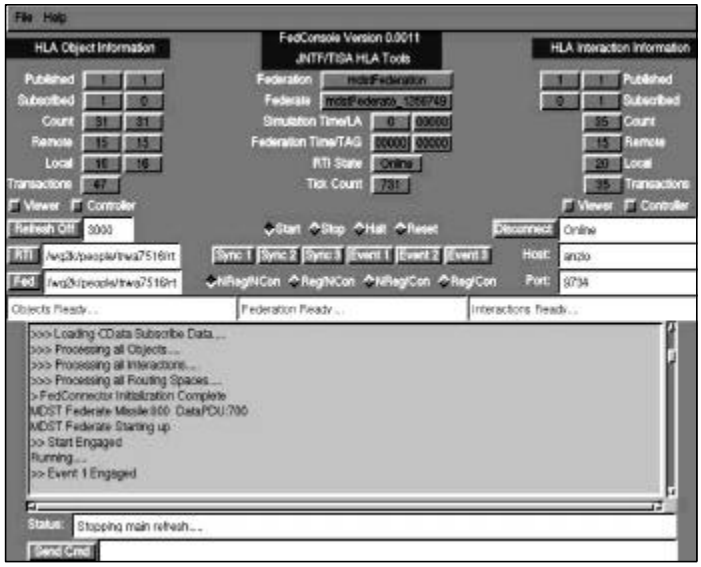

Figure 5.1 FedConsole main screen

FedConsole attaches to FedConnector from anywhere on the network. Because it is written in Java, it can run on virtually any host.

6. Putting it all Together

The driving force behind the development of FedConnector and FedConsole was the HLA Gateway project for Wargame 2000 as shown in Figure 6.1. The federate can adapt to changes to the *.par* or *.fed* file with simple modifications to the definitions and maps.

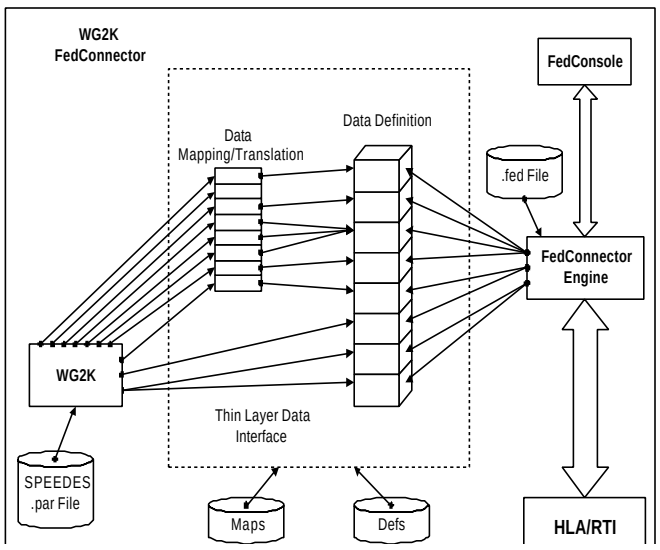

Figure 6.1 The TLDI with local host WG2K

6.1 An Example

A simple federate was constructed using the definition file shown in Figure 3.2 and the source listing shown in Figure 6.2 and 6.3. This federate consisted of one HLA object and one interaction and can federate with another instance of itself while being controlled by FedConsole. The missile object used in this example is a mapping of the MunitionsEntity HLA object. The FedConsole "Start", "Stop" and "Quit" buttons start, stop and quit the main loop. Pressing the Event1 button sent a message and an Event2 creates a missile. All message and missile values are randomly generated. When a missile or message is received it is printed out by the registered update callback function. This code example shown contains a complete test federate.

```
001 //*****************************************************
002 // TestFed  Example      -      Listing one
003 //*****************************************************
004
005 #include "CDTypes.H>                                  // TLDI
006 #include "FedConnector.H"                            // FedConnector
007
008 int run = 0, quit = 0, event1 = 0, event2 = 0; // globals for FedConsole
Cntrl 009
010 int startFed     (char *s, int cmd, int chan) {run = 1;} //  FedConsole
011 int stopFed     (char *s, int cmd, int chan) {run = 0;}   //  Callback
012 int resetFed    (char *s, int cmd, int chan) {quit = 1;}   //  Functions
013 int event1Fed (char *s, int cmd, int chan) {event1++;}  //  for Federate
014 int event2Fed (char *s, int cmd, int chan) {event2++}    //  Control
015
016 int receiveMessage(int type, CData* d, int cnt) {// Message Update
017   int type, size;                                          // local Varibles
018   CByte *dataPtr;
019   d->get("type", &type);                                  // Message.type
accessor
020   d->get("size", &size);                                  // Message.size
accessor
021   d->get("data", (CByte *) &dataPtr);                     // Message.data
accessor
022   cout << "*** Message Data Dump ******** "<< endl;   // Print out the
023   cout << "Type:" << type << " Size: " << size<< endl;      //  Message
data
024   cdata::bufferDump(dataPtr, size);  // Dump Utility
025   cout << "*******************************" << endl;
026   return(CData::DeleteInteraction);  // Delete interaction instance
027 }
028
029 int receiveMissile(int type, CData* d, int cnt) {    // Missile Update
Function
030   float id;                                 // local varibles
031  double mX, mY, mZ;
032   d->get("missileId", &id);   // Obtain Missile ID
033   d->get("mX", &mX);          // Obtain X
034   d->get("mY", &mY);          // Obtain Y
035   d->get("mZ", &mZ);          // Obtain Z
036   cout << "*** Missile Data Dump ******** "<< endl;  // Print out the
037   cout << "ID:" << id << " X: " << mX << " Y: "           // Missile
Data
038           << mY     <<  " mZ:"          << mZ <<     endl;
039   cout << "*******************************" << endl;
040   return(Cdata::InsertInWareHouse);                       // Insert Object
041 }
042
043 int deleteMissile(int type, CData *d, int count) { // Missile Delete function
044  int id;
045  d->get("missileId", &id);                               // Obtain the id
```

Figure 6.2 Example Source Listing 1

The important points from listing 1:

- Lines 5-6 contain the two include files required in the federate source.
- Lines 10-14 provide the callback functions that can be tied to FedConsole button.
- Lines16-27 are the callback function for a message interaction. All we do here is use the built-in accessor functions to print out the message data.
- Lines 29-42 contain the new/update callback function for a missile. Again, using the accessor function the missile data is printed out.
- Lines 44-48 contain the delete callback for a missile object.

```
051 //*****************************************************
052 // TestFed  Example        -   Listing two
053 //*****************************************************
054
055 int main(void) {
056   int lEvent1 = 0, lEvent2=0;                          // Local event values
057   CDTypes dts("TestFed");                              //  CDTypes Object (TLDI)
058   dts.load("TestFed.Def");                             // Load Def's & Maps
059   dts.setUpdate("Missile", receiveMissile);            // Register Missile Update
060   dts.setDelete("Missile",  deleteMissile);            // Register Missile Delete
061   dts.setUpdate("Message", receiveMessage);       // Register Message Update
062   int missile   = dts.find("Missile");                 // Obtain Missile Handle
063   int message = dts.find("Message");                   // Obtain Message Handle
064   FedConnector *f = new FedConnector("TestFed", &dts);    //  New FedConnector
065    f->setStartCB(startFed);                            // Register  the FedConsole Start
Callback
066    f->setStopCB(stopFed);                             // Register  the FedConsole Stop
Callback
067    f->setResetCB(haltFed);                            // Register  the FedConsole Halt
Callback
068    f->setEvent1CB(Event1Fed);                         // Register  the FedConsole Event 1
Callback
069    f->setEvent2CB(Event2Fed);                         // Register  the FedConsole Event 2
Callback
070    f->notes("Test Federate Starting");    // Write string to FedConsole notes area
071    while(!halt) {                                       //  Run till halted
072      f->tickRTI(0.2, 0.5);                              // Tick the RTI
073      if(run) {                                          // If the federate is
Running
074        if(event1>lEvent1) {                             // Create/Send Message
075          int type = rand()%128;                         // Load type with random
value
076          int size = rand()%256;                         //  Load size with random
value
077          CByte *cPtr = new CByte[size];                 // Create a random size data
buffer
078          For(int iSize=0;iSize<size;iSize++) cPtr[iSize]=rand()255;  // Load buffer
079          CData  * mes = dts(message);                   // Create a message instance
080          mes->set("type", type);                        // Set Message.type
081          mes->set("size", size);                        // Set Message.size
082          mes->set("data", cPtr,size);                   // Set Message.data
083          mes->update();                                 // Update message  interaction
084          lEvent1 = event1;                              // advance local event 1 to
FedConsole
085        }                                                // End of Event 1 – Create/Send
Message
086        if(event2>lEvent2) {                             // Create/Send Missile
087          double x = rand()%360;                         // Load x with random value
088          double y = rand()%360;                         // Load y with random value
089          double z = rand()%360;                         // Load z with random value
090          Cdata *mis = dts(missile);                     // Create the missle object
091          mis->set("id", rand()%256);                    // Set Missile.id with random value
092          mis->set("mX", x);                             // Set Missile.mX
093          mis->set("mY", y); );                          // Set Missile.mY
094          mis->set("mZ", z); );                          // Set Missile.mZ
```

Figure 6.2 Example Source Listing 2

The important points from listing 2:

- Line 57 declares the TLDI CDTypes instance.
- Line 58 loads the definition file. An environmental variable can be used to automate this process supporting multiple configurations.
- Lines 59-61 register the update and delete functions.
- Lines 62-63 query the TLDI for the message and missile handles.
- Line 64 declares the FedConnector instance passing the address of the TLDI. The constructor for FedConnector opens the .fed file and correlates it to the TLDI definitions and maps then join the federation.
- Lines 65-69 register the control functions from listing one with FedConnector for use with FedConsole.
- Line 70 sends a string to the FedConsole notes sections.
- Lines 70-101 show the federate main loop.
- Line 72 Ticks the RTI.
- Lines 73-99 provide a run loop that is controlled by the FedConsole "Start" and "Stop" buttons.
- Lines 74-84 send a Message interaction when the "Event1" button is pressed on FedConsole.
  - Lines 75-78 create the random message data.
  - Line 79 creates the message interaction CData instance.
  - Line 80-82 loads the random message data into the message.
  - Line 83 updates the message reflecting the new values.
- Lines 86-96 create Missile Object when the "Event2" is pressed on FedConsole.
  - Lines 87-89 create the random Missile data.
  - Line 90 creates the Missile CData Object.
  - Lines 91-94 load the random missile data.
  - Line 95 updates the missile class reflecting the new values to the RTI.
- Line 101 deletes the FedConnector instance calling the destructor for the fedConnector

object. This will resign the Federate from the federation and cleanup all data.

This test federate example touches on some of the main features of FedConnector and FedConsole. This example allows a federate to connect to a federation, publish and subscribe to one interaction and one mapped object. Control of the example is managed via FedConsole.

7. Conclusion

FedConnector and FedConsole will continue to evolve in support of WG2K and MDST. This evolution will remain generic and general purpose to support rapid HLA federate development.

8. References

[1] Mitch Peckham: “High Level Architecture Gateway for Wargame 2000”, Simulation Interoperability Workshop 99F_SIW_140.

Author Biographies

CHRISTOPHER L. MARTINEZ is a Senior Software Engineer at the Joint National Test Facility, Schreiver AFB, CO. He is a member of the Technology Insertion Studies and Analysis (TISA) group where he has been supporting the War Game 2000 HLA efforts.